

Chapter 1

Object Oriented Programming

1.1 Object oriented approach

Definition 1.1.1: Object Oriented Programming

Object oriented programming (OOP) is a imperative programming paradigm where instructions are grouped together with the part of the state they operate on.

The main **structural components** of all systems are

- Objects: Object is something that takes up space in the real or conceptual world with which somebody may do things. An object is an instance of a class. An object has three main characteristics:
 - Name (or ID): Unique identifier for the object.
 - State: Attributes that describe the object.
 - Operations (or behavior): Actions that the object can perform.
- Class objects: Class is the blueprint of an object.

Main **characteristics** of the approach are

- Encapsulation: The mechanism of hiding the implementation of the object. Only the necessary details are exposed to the user.
- Abstraction: A principle which consists of ignoring the aspects of a subject that is not relevant for the present purpose. It is the process of simplifying complex systems by modeling classes based on the essential properties and behaviors an object must have.
- Inheritance: Mechanism by which one class can inherit attributes and methods from another class. Making new classes from existing ones.
- Polymorphism: Ability to present the same interface for different data types. Same function name but different signatures being used for different types.

1.2 Class

Definition 1.2.1: Class Diagram

A class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

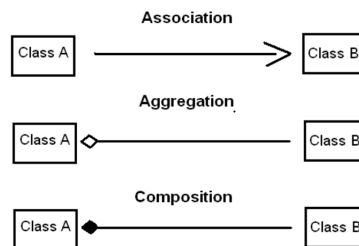
Elements of a **class**:

- Class Name (ID Class): The name of the class. Nouns associated with the textual description of a problem. Singular.

- **Attributes:** The properties or characteristics of the class. Types include Real, Integer, Text, Boolean, Enumerated, etc.
- **Operations (Methods):** The functions or operations that can be performed on the class. Behaviors of the class.

Relations between classes:

- **Association:** A relationship between two classes that indicates how objects of one class are connected to objects of another class.
- **Aggregation:** A special type of association that represents a "whole-part" relationship between classes.
- **Composition:** A stronger form of aggregation that implies ownership and a whole-part relationship where the part cannot exist independently of the whole.
- **Generalization:** A relationship between a more general class (superclass) and a more specific class (subclass) that indicates inheritance.



1.2.1 Class and Method in Python

```

1  class Person:
2      pass # An empty class definition
3
4  p = Person()
5  print(p) # Output: <__main__.Person object at 0x...>

```

Listing 1.1: Empty class definition in Python

```

1  """Define class with method"""
2  class Person:
3      def speak(self):
4          print("Hello!")
5
6  """Create object and call method"""
7  p = Person()
8  p.speak() # Output: Hello!

```

Listing 1.2: Method

```

1  """
2  The __init__ method is a special method in Python classes.
3  It is a method that Python calls when you create a new instance of this class.
4  """
5  class Person:
6      def __init__(self, name):
7          self.name = name # Initialize the name attribute
8      def speak(self):
9          print('Hello, my name is', self.name)
10
11  p = Person('Carlos')
12  p.speak() # Output: Hello, my name is Carlos

```

Listing 1.3: Attributes and __init__ method

Note:

The first argument of every class method, including `init`, is always a reference to the current instance of the class. By convention, this argument is always named `self`.

```

1  class Pet(object):
2      def __init__(self, name, species):
3          self.name = name
4          self.species = species
5      def getName(self):
6          return self.name
7      def getSpecies(self):
8          return self.species
9      def __str__(self):
10         return "%s is a %s" % (self.name, self.species)

```

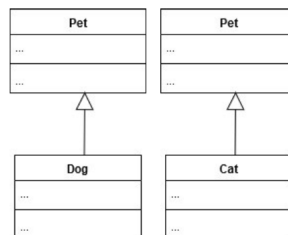
Listing 1.4: Class Pet with attributes and methods

```

1  class Dog(Pet):
2
3      """Dog inherits from Pet"""
4
5      def __init__(self, name, chases_cats):
6          # Call the parent class (Pet) constructor
7          Pet.__init__(self, name, "Dog")
8          self.chases_cats = chases_cats
9
10         def chasesCats(self):
11             return self.chases_cats
12
13     class Cat(Pet):
14
15         """Cat inherits from Pet"""
16
17         def __init__(self, name, hates_dogs):
18             # Call the parent class (Pet) constructor
19             Pet.__init__(self, name, "Cat")
20             self.hates_dogs = hates_dogs
21
22         def hatesDogs(self):
23             return self.hates_dogs

```

Listing 1.5: Inheritance



Example 1.2.1 (Inheritance)

```

myPet = Pet("Boby", "Dog")
myDog = Dog("Rex", True)

```

- `isinstance(myDog, Pet)` # Returns: True
- `isinstance(myDog, Dog)` # Returns: True

- `isinstance(myPet, Pet)` # Returns: True
- `isinstance(myPet, Dog)` # Returns: False

1.2.2 Access Modifiers

There are three types of access modifiers in Python:

- Public: Members (attributes and methods) declared as public are accessible from anywhere.
- Protected: Members declared as protected are accessible within the class and its subclasses. In Python, this is indicated by a single underscore prefix (e.g., `_attribute`).
- Private: Members declared as private are accessible only within the class itself. In Python, this is indicated by a double underscore prefix (e.g., `__attribute`).

```
1 class Person:
2     def __init__(self, name, age):
3         self.__name = name      # Private attribute
4         self.__age = age        # Private attribute
5
6 p = Person("David", 23)
7 p.__name # This will raise an AttributeError
```

Listing 1.6: Private Attributes

```
1 class Person:
2     def __init__(self, name, age):
3         self.name = name       # Public attribute
4         self.age = age         # Public attribute
5
6 p = Person("David", 23)
7 p.name # This will work fine and return "David"
```

Listing 1.7: Public Attributes

```
1 class Person:
2     def __init__(self, name, age):
3         self._name = name      # Protected attribute
4         self._age = age        # Protected attribute
5
6 p = Person("David", 23)
7 p._name # This will work, but it's discouraged to access protected members directly
```

Listing 1.8: Protected Attributes

```
1 class Person:
2     def __init__(self, money=0, energy=100):
3         self.money = money
4         self.energy = energy
5
6     def work(self, hours):
7         if self.energy >= hours * 10:
8             self.money += hours * 10 # Assume earning $10 per hour of work
9             self.energy -= hours * 10
10            print(f"Worked for {hours} hours. Money increased to ${self.money}, energy
11                  decreased to {self.energy}.")
12        else:
13            print("Not enough energy to work.")
```

Listing 1.9: Example Usage

Chapter 2

Data Analysis and Statistics

2.1 Numpy

Definition 2.1.1: Numpy

Numpy is a fundamental package for scientific computing in Python. It provides support for arrays, matrices, and many mathematical functions to operate on these data structures efficiently.

2.1.1 Shape, rank, and size

```
1 c = np.array([1, 2, 3, 4]) # Create a 1D array
2 print(type(c)) # Output: <class 'numpy.ndarray'>
3
4 b = np.array([[1, 2, 3], [4, 5, 6]]) # Create a 2D array
5 shape = b.shape # Output: 2, 3
6 rank = np.ndim(b) # Output: 2
7 size = b.size # Output: 6
```

Listing 2.1: Shape, rank, and size

2.1.2 Manipulating arrays

Reshaping arrays means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping, we can add or remove dimensions or change number of elements in each dimension.

```
1 arr = np.array([4, 4, 3, 4, 5, 6, 7, 9, 0, 10, 17, 12])
2 newarr = arr.reshape(3, 4) # Reshape to 3 rows and 4 columns
3 print(newarr)
4 # Output:
5 # [[ 4  4  3  4]
6 #  [ 5  6  7  9]
7 #  [ 0 10 17 12]]
```

Listing 2.2: Reshaping an array

We can also access an array element and change value to an array.

```
1 a = np.array([10, 20, 30, 40, 50])
2 a[2] = 50 # Change the third element to 50
3 print(a) # Output: [10 20 50 40 50]
```

Listing 2.3: Accessing and changing array elements

```

1 a = np.zeros((2,2)) # Create a 2x2 array of zeros
2 b = np.ones((3,3)) # Create a 3x3 array of ones
3 c = np.full((2,3), 7) # Create a 2x3 array filled with 7s
4 d = np.eye(3) # Create a 3x3 identity matrix

```

Listing 2.4: Creating special arrays

Joining NumPy arrays means putting contents of two or more arrays into a single array.

```

1 a = np.array([1, 2, 3])
2 b = np.array([4, 5, 6])
3
4 c = np.concatenate((a, b)) # Join a and b
5 print(c) # Output: [1 2 3 4 5 6]
6
7 d = np.stack((a, b), axis=0) # Stack a and b vertically
8 print(d)
9 # Output:
10 # [[1 2 3]
11 #  [4 5 6]]
12
13 e = np.stack((a, b), axis=1) # Stack a and b horizontally
14 print(e)
15 # Output:
16 # [[1 4]
17 #  [2 5]
18 #  [3 6]]

```

Listing 2.5: Joining NumPy arrays

```

1 a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
2 print(a)
3 # Output:
4 # [[ 1  2  3  4]
5 #  [ 5  6  7  8]
6 #  [ 9 10 11 12]]
7
8 b = a[:3, 1:3] # Slicing rows 0-2 and columns 1-2
9 print(b)
10 # Output:
11 # [[ 2  3]
12 #  [ 6  7]
13 #  [10 11]]

```

Listing 2.6: Slicing NumPy arrays

Problem 2.1: Slicing arrays

`b[0, 0] = 20`

What happens to array `a`?

Answer: Array `a` is also changed because `b` is a view of `a`.

Problem 2.2: Slicing arrays

How to get a copy of a slice of an array so that changes to the slice do not affect the original array?

Answer: Use the `.copy()` method to create a copy of the slice.

```

1 b = a[:3, 1:3].copy() # Create a copy of the slice
2 b[0, 0] = 20 # Change the first element of b

```

```

3 print(a) # Array a remains unchanged
4 # Output:
5 # [[ 1  2  3  4]
6 #   [ 5  6  7  8]
7 #   [ 9 10 11 12]]

```

Listing 2.7: Creating a copy of a slice

2.1.3 Handling indexes

```

1 a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
2 print(a)
3 # Output:
4 # [[ 1  2  3]
5 #   [ 4  5  6]
6 #   [ 7  8  9]
7 #   [10 11 12]]
8
9 b = np.array([0, 2, 0, 1])
10 print(a[np.arange(4), b]) # Access elements using array of indices
11 # Output: [ 1  6  7 11]
12
13 a[np.arange(4), b] += 10 # Increment selected elements by 10
14 print(a)
15 # Output:
16 # [[11  2  3]
17 #   [ 4  5 16]
18 #   [17  8  9]
19 #   [10 21 12]]

```

Listing 2.8: Fancy indexing

```

1 b = 1
2 b += 10
3 print(b) # Output: 11
4
5 b = 1
6 b = b+10
7 print(b) # Output: 11

```

Listing 2.9: Boolean indexing

```

1 grades = np.array([14, 12, 13, 14, 15, 14, 14])
2 x = np.where(grades == 14) # Find indices where grades are 14
3 print(x) # Output: (array([0, 3, 5, 6]),)

```

Listing 2.10: Searching arrays

2.1.4 Random

Definition 2.1.2: Random

Random number does not mean a different number each time. Random means something that cannot be predicted logically.

```

1 from numpy import random
2
3 # Generating random numbers within an array
4 e = np.random.random((4, 4)) # Create a 4x4 array of random floats in [0.0, 1.0)

```

```

5 # Generating random numbers
6 x = random.randint(100) # Generate a random integer from 0 to 99
7
8 # Generating arrays of random numbers
9 x = random.randint(100, size=(5)) # Generate an array of 5 random integers from 0 to 99
10 x = random.randint(100, size=(3, 4)) # Generate a 3x4 array of random integers from 0
11 to 99
12 x = random.rand(5) # Generate an array of 5 random floats in [0.0, 1.0)

```

Listing 2.11: Generating random numbers

Definition 2.1.3: Random data distribution

A random distribution is a set of random numbers that follow a certain probability density function (PDF).

- Normal distribution (`random.normal`)
 - loc: (mean) where the peak of the bell curve is located
 - scale: (standard deviation) how flat and wide the bell curve is
 - size: output shape
- Binomial distribution (`random.binomial`)
 - n: number of trials
 - p: probability of success on each trial
 - size: output shape

```

1 # Normal Distribution
2 x = random.normal(loc=0.0, scale=1.0, size=(2,3)) # Generate a 2x3 array of random
   numbers from a normal distribution
3 print(x)
4 # Output:
5 # [[ 0.49671415 -0.1382643  0.64768854]
6 # [ 1.52302986 -0.23415337 -0.23413696]]
7
8 # Binomial Distribution
9 x = random.binomial(n=10, p=0.5, size=(3,4)) # Generate a 3x4 array of random numbers
   from a binomial distribution
10 print(x)
11 # Output:
12 # [[5 6 4 3]
13 # [4 7 6 4]
14 # [6 3 5 4]]

```

Listing 2.12: Generating random numbers from different distributions

2.2 Pandas

Definition 2.2.1: DataFrame

Labelled data structure with columns of potentially different types. Like a spreadsheet or SQL table, or a dict of Series objects.


```

1 d = {
2     'col1': [1,2,1,3,1,2],
3     'col2': [1,2,3,4,5,6]
4 }
5 df = pd.DataFrame(data=d)
6
7 df.count() # Count non-NA cells for each column
8 df['col1'].value_counts() # Count unique values in col1
9 df['col1'][1] = 5 # Change value in col1 at index 1 to 5

```

Listing 2.13: Creating and manipulating a DataFrame

Problem 2.3: Copy DataFrames

```
col1 = df['col1'] # Accessing a single column
col1[2] = 99
What is the result in col1 and df?
Answer: Both col1 and df are changed because col1 is a view of df.
```

Problem 2.4: Copy DataFrames

```
new_col1 = col1.copy()
new_col[2] = 9999
What is the result in new_col1 and df?
Answer: Only new_col1 is changed, df remains unchanged because new_col1 is a copy of col1.
```

```

1 df = pd.read_csv('data.csv') # Read data from a CSV file into a DataFrame
2 df.to_csv('output.csv') # Write DataFrame to a CSV file

```

Listing 2.14: Read and save

```

1 df.head() # Display the first 5 rows of the DataFrames
2 df.info() # Display a summary of the DataFrame
3 df.describe() # Generate descriptive statistics
4 df.columns # List all column names

```

Listing 2.15: DataFrame information

```

1 df.at(2, 'col1') # Access a single value for a row/column label pair
2 df.iloc[2, 0] # Access a single value for a row/column pair by integer position
3 df.xs(2) # Returns cross-section (row) at index 2
4 df.loc[:, 'col1'] # Access a group of rows and columns by labels or a boolean array

```

Listing 2.16: Accessing DataFrame elements

Example 2.2.1 (Access to row and columns)

- Cells: `df.iloc[195][0]`
- Rows: `df.iloc[[195], :]`
- Columns: `df.loc[:, 'col1']`

```

1 df1=df2 # Assignment
2 copydf = df.copy(deep=False) # Shallow copy
3 copydf = df.copy(deep=True) # Deep copy

```

Listing 2.17: Copying DataFrames

```

1  # Convert column to numeric, setting errors to NaN
2  df['GDP'] = pd.to_numeric(df['GDP'], errors='coerce')
3
4  # Suppose you have , instead of . in the numbers
5  df['GDP'] = df['GDP'].str.replace(',', '.')
6
7  # Suppose you have $ or € symbols in the numbers
8  df['GDP'] = df['GDP'].str.replace('$', '', regex=True).str.replace('€', '', regex=True)
9
10 # Drop rows with any NaN values
11 df.dropna(inplace=True)

```

Listing 2.18: Data cleaning and conversion

```

1  X.mean() # Calculate mean of each column
2  X.median() # Calculate median of each column
3  X.mode() # Calculate mode of each column
4  X.std() # Calculate standard deviation of each column
5  X.var() # Calculate variance of each column
6  X.corr() # Calculate correlation matrix
7  X.cov() # Calculate covariance matrix
8  X.max() # Calculate maximum of each column
9  X.min() # Calculate minimum of each column
10 X.kurt() # Calculate kurtosis of each column
11 X.skew() # Calculate skewness of each column

```

Listing 2.19: Statistical methods

2.3 Statistics

2.3.1 Statistics module

For averages and measures of central location, we have the following functions in the statistics module:

- `mean()`: Calculate the arithmetic mean of a list of numbers.
- `harmonic_mean()`: Calculate the harmonic mean of a list of numbers.
- `median()`: Calculate the median (middle value) of a list of numbers.
- `median_low()`: Calculate the low median of a list of numbers.
- `median_high()`: Calculate the high median of a list of numbers.
- `mode()`: Calculate the mode (most common value) of a list of numbers.
- `stdev()`: Calculate the standard deviation of a list of numbers.
- `variance()`: Calculate the variance of a list of numbers.
- `pstdev()`: Calculate the population standard deviation of a list of numbers.
- `pvariance()`: Calculate the population variance of a list of numbers.

There are also other packages that could be used, such as NumPy, Pandas, and SciPy, which provide more advanced statistical functions and methods for data analysis.

2.3.2 Statistical tests

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 mu, sigma = 0, 0.1 # mean and standard deviation
5 a = np.random.normal(mu, sigma, 1000) # Generate 1000 random numbers from a normal
    distribution
```

Listing 2.20: Generating random numbers from a normal distribution

```
1 from scipy import stats
2
3 stat, p = stats.normaltest(a) # Perform D'Agostino and Pearson's test for normality
4
5 if p < 0.05:
6     print("The data is not normally distributed")
7 else:
8     print("The data is normally distributed")
9
10 stat, p = stats.shapiro(a) # Perform Shapiro-Wilk test for normality
11
12 if p < 0.05:
13     print("The data is not normally distributed")
14 else:
15     print("The data is normally distributed")
```

Listing 2.21: Normality tests

Definition 2.3.1: Student's t-test

This is a two-sided test for the null hypothesis that 2 independent samples have identical average (expected) values.

```
1 from scipy import stats
2
3 # Generate two independent samples
4 group1 = np.random.normal(loc=0, scale=1, size=100) # Sample from group 1
5 group2 = np.random.normal(loc=0.5, scale=1, size=100) # Sample from group 2
6
7 # Perform the t-test
8 stat, p = stats.ttest_ind(group1, group2, equal_var=False) # Use Welch's t-test for
    unequal variances
9
10 if p < 0.05:
11     print("The means of the two groups are significantly different")
12 else:
13     print("The means of the two groups are not significantly different")
```

Listing 2.22: Student's t-test

Definition 2.3.2: Paired student's t-test

This is a two-sided test for the null hypothesis that 2 related or repeated samples have identical average (expected) values.

```
1 from scipy import stats
2
3 # Generate two related samples
4 before = np.random.normal(loc=0, scale=1, size=100) # Sample before treatment
```

```

5  after = before + np.random.normal(loc=0.5, scale=1, size=100) # Sample after treatment
   (related to before)
6
7  # Perform the paired t-test
8  stat, p = stats.ttest_rel(before, after)
9
10 if p < 0.05:
11     print("The means of the two related groups are significantly different")
12 else:
13     print("The means of the two related groups are not significantly different")

```

Listing 2.23: Paired Student's t-test

Definition 2.3.3: Analysis of Variance Test (ANOVA)

The one-way ANOVA tests the null hypothesis that two or more groups have the same population mean. It is an extension of the t-test for more than two groups, possibly with different sample sizes.

```

1  from scipy import stats
2
3  # Generate three independent samples
4  group1 = np.random.normal(loc=0, scale=1, size=100) # Sample from group 1
5  group2 = np.random.normal(loc=0.5, scale=1, size=100) # Sample from group 2
6  group3 = np.random.normal(loc=1, scale=1, size=100) # Sample from group 3
7
8  # Perform the one-way ANOVA test
9  stat, p = stats.f_oneway(group1, group2, group3)
10
11 if p < 0.05:
12     print("At least one group mean is significantly different from the others")
13 else:
14     print("All group means are not significantly different from each other")

```

Listing 2.24: ANOVA

2.4 Time Series

The purpose of time series analysis is to understand the underlying structure and function that produced the observed data, and to fit a model and make forecasts.

To decompose time series data, we can use the `seasonal_decompose` function from the `statsmodels` library, which decomposes a time series into its trend, seasonal, and residual components.

We can also plot the autocorrelation function (ACF) and partial autocorrelation function (PACF) to identify the presence of autocorrelation in the data, which can help us determine the appropriate order of an ARIMA model for forecasting.

The ARIMA (AutoRegressive Integrated Moving Average) model is a popular method for forecasting time series data, which combines autoregressive (AR) and moving average (MA) components, along with differencing to make the data stationary.

After fitting the model, we can check the residuals to ensure that they are normally distributed and do not exhibit autocorrelation, which would indicate that the model is a good fit for the data.

We can also train and test the data by splitting the time series into a training set and a test set, fitting the model on the training data, and evaluating its performance on the test data using metrics such as mean absolute error (MAE) or root mean squared error (RMSE).

```

1  import statsmodels.tsa.api as tsa
2  import pandas as pd
3  import matplotlib.pyplot as plt
4  from pandas.plotting import autocorrelation_plot
5  import statsmodels.tsa.api as tsa

```

```

6 from math import sqrt
7 from sklearn.metrics import mean_squared_error
8
9 # Load time series data
10 df = pd.read_csv('tourismPortugal.csv', sep=';')
11 df['month'] = pd.to_datetime(df['month'])
12 df = df.set_index(df['month'])
13 df = df.drop(['month'], axis=1)
14
15 # Decompose the time series into trend, seasonal, and residual components
16 result = ts.seasonal_decompose(df, model='multiplicative', period=12)
17
18 figure = result.plot() # Plot the decomposed components
19
20 # Plot the autocorrelation function (ACF) and partial autocorrelation function (PACF)
21 serie = df[['tourists', 'month']]
22 autocorrelation_plot(serie) # Plot the ACF
23 plt.show()
24
25 # Fit an ARIMA model to the data
26 serie = serie.set_index('month')
27 serie.index = serie.index.to_period('M') # Convert index to monthly periods
28
29 p = 4 # Order of the autoregressive part
30 d = 1 # Degree of differencing
31 q = 0 # Order of the moving average part
32
33 model = ts.arima.ARIMA(serie, order=(p, d, q))
34 results = model.fit()
35 print(results.summary()) # Print the model summary
36
37 # Check the residuals
38 residuals = pd.DataFrame(results.resid)
39 residuals.plot(title="Residuals") # Plot the residuals
40 plt.show()
41
42 residuals.plot(kind='kde', title='Density of Residuals') # Plot the density of
43     residuals
44 plt.show()
45
46 print(residuals.describe()) # Print the summary statistics of residuals
47
48 # Train and test the model
49 X = serie.values
50 size = int(len(X) * 0.70) # Split the data into training and test sets
51 train, test = X[0:size], X[size:len(X)]
52 history = [x for x in train]
53 predictions = list()
54
55 # Walk-forward validation
56 for t in range(len(test)):
57     model = ts.arima.ARIMA(history, order=(p, d, q))
58     model_fit = model.fit()
59     output = model_fit.forecast()
60     yhat = output[0]
61     predictions.append(yhat)
62     obs = test[t]
63     history.append(obs)
64     print('predicted=%f, expected=%f' % (yhat, obs))
65
66 # Evaluate forecasts

```

```

66 rmse = sqrt(mean_squared_error(test, predictions))
67 print("Test RMSE: %.3f", rmse)
68
69 # Plot the actual vs predicted values
70 plt.plot(test)
71 plt.plot(predictions, color='red')
72 plt.show()

```

Listing 2.25: Time series analysis and forecasting

Other approaches such as LSTM, Prophet, and GARCH models can also be used for time series forecasting, especially when dealing with non-linear patterns or volatility in the data.

2.5 Regression

The main statsmodels API is split into models:

- `statsmodels.api`: The main API for statistical models, including linear regression, generalized linear models, and time series analysis. Cross-sectional models and methods.
- `statsmodels.formula.api`: Provides a formula interface for specifying statistical models using R-style formulas. A convenience interface for specifying models using formula strings and DataFrames.
- `statsmodels.tsa.api`: Contains tools for time series analysis, including ARIMA models, seasonal decomposition, and forecasting. Time-series models and methods.
- `statsmodels.graphics.api`: Provides functions for visualizing statistical models and results, such as residual plots and influence plots.

2.5.1 Linear regression

Linear models with independently and identically distributed errors, and for errors with heteroscedasticity or autocorrelation of unknown form.

- **OLS (Ordinary Least Squares)**: A linear regression model that estimates the relationship between a dependent variable and one or more independent variables by minimizing the sum of squared residuals. Assumptions: linearity, independence, homoscedasticity, normality of errors, and no multicollinearity.
- **GLM (Generalized Linear Models)**: A flexible generalization of linear regression that allows for response variables that have error distribution models other than a normal distribution. It includes logistic regression, Poisson regression, and others. Assumptions: the response variable follows an exponential family distribution, the link function is correctly specified, and the observations are independent.
- **GEE (Generalized Estimating Equations)**: A method for estimating the parameters of a generalized linear model with possible correlation between outcomes. It is used for longitudinal data analysis and clustered data. Assumptions: the correlation structure of the data is correctly specified, and the observations are independent across clusters. Instead of assuming independent observations, GEE accounts for within-subject correlation.
- **GAM (Generalized Additive Models)**: A generalization of linear models that allows for non-linear relationships between the dependent and independent variables by using smooth functions. It is used for modeling complex relationships in data. Assumptions: the response variable follows an exponential family distribution, the link function is correctly specified, and the observations are independent. GAMs do not assume a specific functional form for the relationship between the dependent and independent variables, allowing for more flexibility in modeling non-linear relationships.
- **Robust linear models**: These models are designed to be less sensitive to outliers and violations of assumptions than traditional linear regression models. They include methods such as Huber regression and RANSAC (Random Sample Consensus). Assumptions: Robust linear models do not rely on the same assumptions as traditional linear regression models, such as normality of errors or homoscedasticity. Instead, they are designed to be more flexible and robust to violations of these assumptions, making them suitable for data with outliers or non-normal error distributions.

- LMM (Linear Mixed Models): These models are used for data that have both fixed effects (parameters associated with an entire population) and random effects (parameters associated with individual experimental units). They are commonly used in fields such as psychology, ecology, and medicine for analyzing data with hierarchical or nested structures. Assumptions: linearity, normality of residuals, homoscedasticity, and independence of observations. Additionally, LMMs assume that the random effects are normally distributed and that the fixed effects are correctly specified. LMMs are particularly useful for analyzing data with repeated measures or clustered data, where observations are not independent, as they can account for the correlation between observations within clusters or subjects.

2.5.2 OLS assumptions

OLS assumes a **linear relationship** between the independent variables and the dependent variable. Verification methods include scatter plots to visualize the relationship between independent variables and the dependent variable. Residual plots by plotting the residuals against the fitted values to check for patterns that indicate non-linearity. And polynomial features by fitting a polynomial regression to see if higher-order terms improve the model fit.

```

1  import numpy as np
2  import pandas as pd
3  import statsmodels.api as sm
4  import matplotlib.pyplot as plt
5  import seaborn as sns
6
7  # Example dataset
8  df = pd.DataFrame({
9      'X': np.linspace(1, 100, 100),
10     'Y': np.linspace(1, 100, 100) + np.random.normal(0, 5, 100)
11 })
12
13 # Fit OLS model
14 X = sm.add_constant(df['X']) # Add intercept
15 model = sm.OLS(df['Y'], X).fit()
16
17 # Scatter plot
18 sns.regplot(x=df['X'], y=df['Y'],
19             scatter_kws={'alpha':0.5},
20             line_kws={"color":"red"})
21 plt.title("Checking Linearity with Scatter Plot")
22 plt.show()
23
24 # Residual plot
25 plt.scatter(
26     model.fittedvalues, model.resid, alpha=0.5)
27 plt.axhline(y=0, color='red', linestyle='--')
28 plt.xlabel("Fitted Values")
29 plt.ylabel("Residuals")
30 plt.title("Residual Plot for Linearity Check")
31 plt.show()

```

Listing 2.26: Checking linearity assumption in OLS

Multicollinearity occurs when two or more independent variables in a regression model are highly correlated, meaning that one can be linearly predicted from the others with a substantial degree of accuracy. This can lead to unstable estimates of regression coefficients and make it difficult to determine the individual effect of each independent variable on the dependent variable. Verification methods include correlation matrix by calculating the correlation coefficients between independent variables to identify pairs that are highly correlated. Variance Inflation Factor (VIF) by calculating the VIF for each independent variable, where a VIF value greater than 5 or 10 indicates a potential multicollinearity problem.

```

1  from statsmodels.stats.outliers_influence import variance_inflation_factor
2
3  # Create feature matrix
4  X = pd.DataFrame({'X1': np.random.rand(100), 'X2': np.random.rand(100) * 2,

```

```

5         'X3': np.random.rand(100) * 3})
6     X = sm.add_constant(X)
7
8     # Compute VIF for each variable
9     vif_values = []
10    for i in range(X.shape[1]):
11        vif = variance_inflation_factor(X.values, i)
12        vif_values.append((X.columns[i], vif))
13
14    # Convert to DataFrame
15    vif_data = pd.DataFrame(vif_values, columns=["Variable", "VIF"])
16    # Display the results
17    print(vif_data)

```

Listing 2.27: Checking multicollinearity using VIF

Residuals should have constant variance, **homoscedasticity**, across all levels of the independent variables. Verification methods include residual plots by plotting the residuals against the fitted values to check for patterns that indicate heteroscedasticity. Breusch-Pagan test by performing the Breusch-Pagan test to statistically assess the presence of heteroscedasticity, where a significant p-value indicates heteroscedasticity.

```

1     import statsmodels.stats.diagnostic as smd
2
3     # Residual plot
4     plt.scatter(model.fittedvalues, model.resid, alpha=0.5)
5     plt.axhline(y=0, color='red', linestyle='--')
6     plt.xlabel("Fitted Values")
7     plt.ylabel("Residuals")
8     plt.title("Checking Homoscedasticity")
9     plt.show()
10
11    # Breusch-Pagan Test
12    bp_test = smd.het_breuschpagan(model.resid,
13                                   model.model.exog)
14    print(f"Breusch-Pagan p-value: {bp_test[1]}")

```

Listing 2.28: Checking homoscedasticity

The residuals should not be correlated. This is important in time series data, where autocorrelation can be a common issue. Verification methods include autocorrelation function (ACF) plot by plotting the ACF of the residuals to check for significant autocorrelation at different lags. Durbin-Watson test by performing the Durbin-Watson test to statistically assess the presence of autocorrelation, where a value close to 2 indicates no autocorrelation, values less than 2 indicate positive autocorrelation, and values greater than 2 indicate negative autocorrelation.

```

1     from statsmodels.stats.stattools import durbin_watson
2
3     dw_stat = durbin_watson(model.resid)
4     print(f"Durbin-Watson Statistic: {dw_stat}")

```

Listing 2.29: Checking autocorrelation of residuals

The residuals should be approximately normally distributed, which is important for the validity of hypothesis tests and confidence intervals in OLS regression. Verification methods include Q-Q plot by creating a Q-Q plot of the residuals to visually assess their normality, where points should approximately follow a straight line if the residuals are normally distributed. Shapiro-Wilk test by performing the Shapiro-Wilk test to statistically assess the normality of the residuals, where a significant p-value indicates that the residuals are not normally distributed. Histogram and KDE plot by plotting a histogram and kernel density estimate (KDE) of the residuals to visually assess their distribution, where a bell-shaped curve suggests normality.

```

1     import scipy.stats as stats
2
3     # Histogram and KDE
4     sns.histplot(model.resid, kde=True, bins=30)

```



```

5 plt.title("Residuals Distribution")
6 plt.show()
7
8 # Q-Q Plot
9 sm.qqplot(model.resid, line="45")
10 plt.title("Q-Q Plot for Residuals")
11 plt.show()
12
13 # Shapiro-Wilk Test
14 shapiro_test = stats.shapiro(model.resid)
15 print(f"Shapiro-Wilk p-value: {shapiro_test.pvalue}")

```

Listing 2.30: Checking normality of residuals

Note:

The `summary()` function in Statsmodels provide important regression output, but it does not directly test OLS assumptions. However, it gives useful clues about potential violation.

- R-squared and adjusted R-squared: indicate model fit but do not test assumptions.
- F-statistics and Prob (F-statistics): tests if at least one predictor is significant.
- Coefficients and p-values: show predictor significance but do not check multicollinearity.
- Standard errors: can be affected by heteroskedasticity.
- Durbin-Watson statistic: helps detect autocorrelation, should be close to 2.
- Omnibus and Jacque-Bera tests: check for normality of residuals, should have a high p-value.

```

1 import statsmodels.api as sm
2
3 model = sm.OLS(y,X).fit()
4 print(model.summary())

```

Listing 2.31: Fitting an OLS model and checking assumptions